py-image-server

A HTTP server which provides flexible image processing functionality.
The HTTP server is realized with fastAPI, running through uvicorn.
The image processing functionalities are implemented with opencv.

The user sends a request in JSON format through the API to the webserver.
The webserver picks up the request in a parallel process and performs the desired image operations.
Images are stored locally on the server, and the request defines the paths on the server.

Contents:

[1] Design of the software

The software consists of two main components:

1. A webserver based on fastAPI that handles HTTP requests in JSON format in concurrent processes
2. An image processor that offers a variety of image modifications

Server design

The webserver is based on the fastAPI package, and is run with Uvicorn.
The requests are defined using the fastAPI JSON format:

```
{
    images:     [{"input":"inputPath1","output":"outputPath1"},
                 {"input":"inputPath2","output":"outputPath2"},
                 ...
                 {"input":"inputPathN","output":"outputPathN"}],
    operations: ["operation1",
                 "operation2",
                 ...
                 "operationN"]
}
```

- input: the path to the image
- output: the output path for the processed image
- operation: processing method name and arguments

An example of a request looks as follows:

```
request_JSON = {
  "images":
[{"input":os.path.join(inputPath,"cyclo1.jpg"),"output":os.path.join(outputPath,"
cyclo1_processed.jpg")},

{"input":os.path.join(inputPath,"cyclo2.jpg"),"output":os.path.join(outputPath,"c
yclo2_processed.PNG")},

{"input":os.path.join(inputPath,"cyclo3.jpg"),"output":os.path.join(outputPath,"c
yclo3_processed.jpg")},

{"input":os.path.join(inputPath,"cyclo4.jpg"),"output":os.path.join(outputPath,"c
yclo4_processed.PNG")}],
  "operations":
["splitQuadrants=True","resize=2","splitQuadrants=True","resize=2"]
}
```

This loads in total 4 images, processes it with the defined operations from left to right: splitting into four quadrants, resizing with scalingfactor 2, splitting all 4 quadrants into 16 images, and again scaling with scale factor two.
The result is an 16 images with unique regions of the base image, with the same size as the original.

Image processing functionalities

There are in total 3 different functionalities in the image processor,
which can be called in any order defined in the request

1. Resizing the image to any reasonable size smaller than the origin image.
2. Split the image into left-top/right-top/left-bottom/right-bottom sub-images and process/store them separately
3. Blur the full image, for example by applying a Gaussian filter or by downsampling followed by upsampling.

The operations can be called through the following example

```
"resize=scalingFactor", with scalingFactor=(0.05,100]
"split=True"
"blur=kernelSize", with kernelSize=[0,inf)
```

Here, the resizing scaling factor is a float that denotes the percentag scaling to be done.
If split in the operations list, it will work for any string besides "None", "false", or "False", which will skip splitting.
The blur kernelSize is an integer that indicates the Gaussian blur kernel size. If this is zero, the kernelSize will be computed automatically from Sigma (https://www.tutorialkart.com/opencv/python/opencv-python-gaussian-image-smoothing/).
If the user sends a request that has a configuration outside the bounds, it skips the operation.

[2] Installation and first run

Preparing Python environment

Start your python command prompt. From here, you have a few options.
Select your preference:

1. (recommended) Create an Anaconda environment from conda_environment.yml:

```
cd path/to/py-image-server
conda env create -f conda_environment.yml
conda activate image-server
```

After you're done with the software, run the following command to clean up:

```
conda remove --name image-server --all
```

2. Create your own virtual environment and install dependencies with PIP:

```
(activate your virtual environment)
cd path/to/py-image-server
pip install -r requirements.txt
```

3. (not recommended) Install dependencies directly through pip with requirements.txt:

   pip install -r requirements.txt

Running the server

The Uvicorn server can be ran by simply executing the main.py script:

```
python main.py
```

This starts a local server at port 8888.
To make sure everything is works, navigate to:

http://localhost:8888/docs#/default/create_request_items__post

Then, run a test by clicking "try it out" > "execute". This should show the following response:

```
{
  "images": [],
  "operations": [
    "string"
  ]
}
```

When you receive this response the server works properly.

[3] Test cases

Main functionality pytest testing

Now you know that the server works, extensive tests can be run by executing the following command:

```
pytest test_main.py --verbose
```

Or for extra debug logging the following command:

```
pytest test_main.py --verbose -s
```

The tests will send requests with images from the folder 'testimages', and store the processed images in 'testoutput'.
In these folders, you can verify that indeed the required processing steps are okay.

Note: At this point the software has only been tested and developed for a request format that exactly matches the defined template.
A production ready version should deal with wrong requests without crashing by checking if it is in the correct format.

Concurrency test

The concurrency is realized through the fastAPI callback definition, which can easily facilitate parallel processing.
The concurrent processing was verified with a concurrency testing mode that can be activated at the top of the file: ***imageprocessor.py***:

```
import os
import time
from typing import List
import cv2
from pydantic import BaseModel

TESTING_CONCURRENCY = False         << SET THIS TO TRUE
...
```

By activating this mode, the processor will add a delay in the image processing
that allows the user to send multiple requests before the request processing finishes.
The user has 30 seconds to spawn an additional process:

```
...
def ProcessRequest(request=Request):
    if TESTING_CONCURRENCY:
        print("In concurrency testing mode. Sleeping for 30 seconds...")
        time.sleep(30)
        ...
```

To perform the concurrency test, do the following:

1. Set **TESTING_CONCURRENCY = True**
2. Run the server through: ***'python main.py'***
3. Open http://localhost:8888/docs#/default/create_request_items__post and **send a first request.**
   In your console you should see a notification that the processing started
4. Open the link again, and **send a second request**. Again you should see the processing starting.
   While the first request is still processing, the server responds to the second request.
   This indicates that it can handle concurrent requests
5. After 30 second you'll see that both processing requests have finished.

Example output:

```
INFO:     Started server process [15144]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://localhost:8888 (Press CTRL+C to quit)
INFO:     127.0.0.1:50598 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:50598 - "GET /openapi.json HTTP/1.1" 200 OK
In concurrency testing mode. Sleeping for 30 seconds...         << FIRST REQUEST
INFO:     127.0.0.1:50601 - "GET /docs HTTP/1.1" 200 OK
INFO:     127.0.0.1:50601 - "GET /openapi.json HTTP/1.1" 200 OK
In concurrency testing mode. Sleeping for 30 seconds...         << SECOND REQUEST
Initialized 0 processors.
Processed 0 images.                                             << FIRST DONE
INFO:     127.0.0.1:50598 - "POST /items/ HTTP/1.1" 200 OK
Initialized 0 processors.
Processed 0 images.                                             << SECOND DONE
INFO:     127.0.0.1:50601 - "POST /items/ HTTP/1.1" 200 OK
```

Here, no processors are actually spawned since the test messages are empty.
However, testing concurrency this is okay.

[4] Development time log

```
1.5 hour research HTTP server, API & Concurrency
5 hours development & testing
1.5 hours documentation & installation steps
```